A Smoothed Upsample Algorithm Using Streaming SIMD Extensions

Version 2.1

01/99

Order Number: 243656-002

Table of Contents

Revision History

Revision	Revision History	Date
2.1	FCS revision.	01/99

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provide floating point single-instruction, multiple-data (SIMD) instructions and provide additional SIMD integer instructions. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, and spatial (3D) audio.

The algorithm presented in this application note is called "smoothed upsample", and is a subset of a more general class called a "resample" algorithm. This paper presents the use of the Streaming SIMD Extensions for implementing the smoothed upsample algorithm, and provides code examples in C, hand-coded Streaming SIMD Extensions assembly, and compiler intrinsics for Streaming SIMD Extensions.

2 The Smoothed Upsample Algorithm

Resampling is a process of transforming a sampled signal. It can be thought of as either changing the duration and frequency of a signal given a fixed sampling rate, or as converting a signal of fixed duration to a new sampling rate. Both an increase or decrease in the number of samples representing the signal can be achieved. The smoothed upsample algorithm presented in this paper concentrates on the process of increasing the number of samples.

The algorithm operates on signed 16-bit input samples, transforms them, and produces a larger number of signed 16-bit output samples. As an example, the sample values might represent digitized audio. The transform can be considered as either a change in the sampling rate of a fixed signal, or a frequency shift of a signal for a fixed sampling rate.

Upsampling is typically applied to a sequence of digital samples taken at some fixed rate from an analog signal. The algorithm attempts to increase the number of samples, in such a way that the resulting sample sequence still accurately represents the original analog signal. One common approach for upsampling is to linearly interpolate between adjacent samples. For example, one would average each pair of samples to create a new sample value between the two original samples.

However, linear interpolation introduces some distortion, if the original sampling was done correctly. An analog signal should be sampled at twice the rate of the highest frequency component of the analog signal. If, the sampling rate is not high enough, the sampled signal would contain "aliasing". For example, an improperly sampled audio signal, when played back, might sound distorted. Linear interpolation between samples effectively introduces high frequency components that should not have existed in the original signal.

Smoothed upsampling attempts to make a better "guess" at the original signal shape, by fitting a smooth curve through four adjacent sample points, and taking new samples only between the center two samples. This should minimize the introduction of false higher frequency components, and better match the original signal shape.

2.1 Applications for a Smoothed Upsample Algorithm

This algorithm could be applied to any sequential sample stream either to increase the number of samples, or as the first step in reducing the number of samples. The latter typically would be done by applying the Smoothed Upsample algorithm, followed by application of a filter to produce a smaller number of samples.

For example, an audio signal originally sampled at 22 kHz could be directly upsampled to a 44 kHz sample rate. Or a 44 kHz signal could be upsampled to an 88 kHz sample rate, and then filtered down to a 22 kHz sample rate.

A video signal might be processed with smoothed upsampling to stretch the width of the video picture, or as the first step toward compacting the picture's width.

2.2 Implementing the Smoothed Upsample Algorithm

The upsample algorithm reads in four sequential 16 bit integral samples, four sets at a time. A weighted output sample value is generated using an equation that generates a smooth curve through the four samples nearest the output sample's position. By choosing output sample positions spaced relatively closer or further apart than the original samples, a frequency shifted output sample stream can be generated. The resulting samples are then converted back to 16 bit signed integral form and stored four at a time.

The algorithm can be chosen to increase the number of samples by any amount – the algorithm ensures a reasonably smooth interpolation between the input samples.

The algorithm derives points on a continuous curve passing through two points B and C. It computes a weighted sum of three linear equations, matching hypothetical lines passing through each of three pairs of samples selected from four adjacent samples A, B, C, and D (in sample order). The weight applied is derived from the relative sampling position – a value between 0.0 and 1.0 representing a sample position between the two middle samples of the four.

The equation used should provide a continuous curve with continuous slope through points B and C when joined with similarly calculated curves for AB and CD.

Let `p` be the fractional position (0 to 1.0) between B and C. Let `q=(1-p)`. Use A,B,C,D to represent the values of samples at those points. Assuming a sample spacing of 1, the slopes between adjacent sample points are B-A, C-B, and D-C.

The equations for the three lines near `p` are

$L = B + p(B-A)$

$M = B + p(C-B)$

$N = C - q(D-C)$

L is given a weight of `q` (its influence declines as one moves from B to C, as q declines). N is given a weight of `p` (its influence increases as one moves from B to C, as `p` increases). M is given constant weight as `p` and `q` vary. The resulting values are summed and divided by 2 to fit the resulting curve through B and C.

$S = (0.5)(q(L) + p(N) + M)$

Expanding this produces:

$S = (0.5)(q(B + p(B-A)) + p(C - q(D-C)) + B +$

p(C-B))

OR

S =B + (0.5 * (p(q(-A+B+C-D) + 2*(C-B)));

2.2.1 Implementation Techniques

Since the original values are signed integers, and the algorithm uses floating point, it is necessary to convert the samples. Two signed sample values are inserted into the high order 16 bits of each of two DWORDS (unsigned 32 bit values) of an MMX™ technology register using the `pinsrw` instruction. The `psrad` (arithmetic shift right DWORD) instruction is then used to move the 16 bits to the low order 16 bits of the DWORDs, with sign extension. Finally, the two values can be converted to floating point in a Pentium® III register using the `cvtpi2ps` instruction.

3 Conclusion

Streaming SIMD Extensions can provide performance improvements for smoothed upsampling, and in general, for any type of "resampling" algorithm.

The performance of the Streaming SIMD Extensions optimized assembly version of the smoothed upsample algorithm can be compared to the intrinsics version of the same algorithm, to the FVEC C++ class library version, or to the C coded version. The assembly version is substantially faster than the C coded version.

4 C Code Example

This C code is presented as an aid to understanding the required functionality, and is not optimized.

```
void    C_Resample(
float fRatio,
short *pIn,
short *pOut,
int nInSize,
int nOutSize )
{
    int        nPosition = 0;    // index of first sample
    int        nOldPos = 0;
    float    fPosition = 0.0;    // FP position
    short    *pBPos;            //  2nd sample position

    // equation components for samples A, B, C, D
    float    ABCD;
    float    CB2;
    short    B;
    float    p = 0.0;   // initial p value
    float    q = 1.0;   // initial q = 1-p value
    float    tmp;
```

```
    // Initial input array element pointer
    pBPos = pIn;    // A is "before" the input
//  sample array, so point at B

    // INITIAL EQUATION COMPONENTS
    // (A=B, so -A+B+C-D = C-D)
    ABCD = (float)( *(pBPos+1) - *(pBPos+2) ) ;
    CB2 = 2.0 * (float)( *(pBPos+1) - *pBPos );
    B    = (*pBPos);

    // for all output samples
    for( int samp = 0; samp < nOutSize; samp++ )
    {
        // THE EQUATION
        tmp = (float)B +
   ( 0.5 * (p * ( (q * ABCD ) + CB2 ) ) );

        if( tmp > (float)((1<<15) -1)  )
        {
            pOut[samp] = (1<<15) -1;
        }
        else if( tmp < (float)(1 - (1<<15))  )
        {
            pOut[samp] = 1 - (1<<15);
        }
        else
            pOut[samp] = (float)tmp;

        // increment position by the ratio of # of
        // in samples to # of out samples
        fPosition += fRatio;
        nPosition = (int)fPosition;
        p = fPosition - (float)nPosition;
        q = (1.0 - p);

        if( nOldPos != nPosition )
        {
            pBPos += (nPosition - nOldPos);
            ABCD = (float)((*pBPos) - *(pBPos-1) +
*(pBPos+1) - *(pBPos+2) ) ;
            CB2 = 2.0 * (float)( *(pBPos+1) –
*pBPos );
            B = (float)(*pBPos);

            nOldPos = nPosition;
        }
    }
}
```

5 Streaming SIMD Extensions Assembly Code Example

The assembly code for the upsample algorithm can be found in `\samples\Upsample\Resample.cpp`. Open the file and search for the string, **ASM**, to locate the beginning of the assembly code sample.

Note that the code is unrolled to do two sets of four values per iteration. This allowed a marginal speed improvement.

The input data is a serial stream, that is sampled at varying positions. This makes it inherently difficult to avoid data cache line splits if more than one word at a time were loaded. To avoid this, the `pinsrw` instruction is used to load samples one at a time, and insert them into an MMX technology register.

The 16 bit signed samples needed to be converted to 32 bit signed samples. This requires that the samples be placed in the high order word of a DWORD (which `pinsrw` can do), and right-shifted with sign extension.

The appended code is unrolled to pursue a total of 8 output samples per iteration.

6 Code Example Using Compiler Intrinsics for Streaming SIMD Extensions Intrinsics And Class Library

The intrinsics and class library (FVEC) code for the upsample algorithm can be found `\samples\Upsample\Resample.cpp`. Open the file and search for the string, **INTRINSICS** or **FVEC**, to locate the beginning of the intrinsics or class library code samples.